Multidiszciplináris tudományok, 8. kötet. (2018) 1 sz. pp. 57-64

KIMENŐ CSOMAGOKHOZ TARTOZÓ PROCESSZEK AZONOSÍTÓJÁNAK NAPLÓZÁSA A LINUX KERNEL NETFILTER KOMPONENSÉBEN

Soós Róbert
hallgató, Miskolci Egyetem, Gépészmérnöki Kar
3515 Miskolc, Miskolc-Egyetemváros, e-mail: totalse95@gmail.com

Absztrakt

Jelen cikk keretében a szerző bemutatja a Linux kernel hálózati struktúráját, a létező módszereket a csomagok megfigyelésére, valamint egy általa készített implementációt, melynek segítségével az IPtables LOG target-jének használata során a csomagot küldő processz azonosítója is bekerül a kernel log-ba. Konkrét példán keresztül is bemutatásra kerül, hogy hogyan segít az implementáció az Apache webszerver PHP scriptjeinek megfigyelésében.

Kulcsszavak: *Linux kernel, netfilter, matchelés, socket mechanizmusok*

Abstract

Within the frame of this paper, the author describes the network structure of the Linux kernel, the existing methods for monitoring packets, and an implementation that he uses to include the ID of the process sending the packet in the kernel log when using the IPtables LOG target. It also provides a concrete example of how the implementation helps monitor PHP scripts on the Apache web server.

Keywords: *Linux kernel, netfilter, matching, socket mechanism*

1. Bevezetés

Napjainkban az informatika szinte bármely területén találkozhatunk Linux alapú rendszerekkel. Az Linux széleskörű elterjedésének számtalan oka van: például egyszerű testreszabhatósága, finomhangolhatósága, ingyenessége, szabad bővíthetősége. A szerver oldalon egyik fő előnye mégis talán kiforrott hálózatkezelésében rejlik. A kernel hálózati szolgáltatásai rendkívül jól használhatók, többek között beépített tűzfallal is rendelkezik, ráadásul rengeteg felhasználói program is létezik a kapcsolatok kezelésére. Ezek viszont többnyire a csomagokra koncentrálódnak, holott gyakran előfordul, hogy azok küldőjéről szeretnénk többet megtudni.

Célszerű lehet Linux rendszereken a hálózat használatának figyelése processzenként. Ha minden kimenő csomagról tudnánk, hogy melyik processz küldi, optimalizálni tudnánk a hálózat terhelését, megfigyelhetnénk, hogy a processzek mekkora forgalmat generálnak, ezzel mennyire terhelik a hálózatot. Az adatok tanulmányozásával kiszűrhetnénk azokat a programokat, amelyek túl sokat használják fölöslegesen a hálózatot, nem kívánatos címre továbbítanak csomagokat, vagy esetleg adatokat szivárogtatnak ki a számítógépről. Ezeket egyszerűen bezárhatnánk, vagy akár a kernel tűzfala segítségével eldobhatnánk az általuk előállított csomagokat.

Ehhez arra lenne szükség, hogy valamilyen módon naplózni tudjuk a kifelé menő csomagok küldő processzeinek process ID-jét, vagyis PID-jét. A következőkben bemutatok egy módszert arra, hogy ezt

a PID-et már a csomag keletkezésekor eltároljuk, majd a kernel log-ban a megfelelő csomag adataival együtt megjelenítsük.

2. Netfilter és IPtables

A Netfilter egy hálózati csomagszűrést és nyomkövetést lehetővé tevő keretrendszer, amely a 2.4-es verzió óta a Linux kernel része. Úgynevezett „hook"-okból (láncokból) áll, amelyeken a hálózati csomagok átmennek, így lekérdezhetjük az adataikat, naplózhatjuk a forgalmat, vagy akár manipulálhatjuk, eldobhatjuk a csomagokat.

A Netfilter funkcióit a felhasználók az IPtables nevű parancssori programmal használhatják, segítségével egyszerűen adhatnak hozzá és törölhetnek szabályokat, melyek a Netfilter alrendszer részeként működnek [7]. A szabályok segítségével lehetővé válik bizonyos feltételeknek megfelelő, „match"-elt csomagok eldobása, új címre küldése, vagy akár adatinak naplózása. A naplózás az IPtables LOG target-jének segítségével történik; így az adatok a kernel log-ba kerülnek. Ezeket a „dmesg" paranccsal olvashatjuk ki legegyszerűbben.

3. Lehetséges részmegoldások

Némi utánajárás után találtam néhány részmegoldást, áthidaló lehetőséget a problémára, melyeket a következőekben mutatok be.

Matchelés process ID alapján

Az IPtables korábbi verziója biztosított a kívánatoshoz hasonló funkciót: az *„owner"* modul használatával rendelkezésre állt a *„--pid-owner (pid)"* valamint a *„--cmd-owner (name)"* kapcsoló [8]. Segítségükkel lehetséges lenne olyan feltételek létrehozása, amely adott process id vagy processz név esetén match-elnek. Bár ez önmagában nem oldaná meg a feladatot, ezen két kapcsoló segítségével létrehozható egy kezdetleges megoldás: bizonyos „gyanús" processzek pid-jének megadásával egyszerűen log-olhatnánk azok kimenő forgalmát. Ezt a következő paranccsal tehetnénk meg:

```
iptables -A OUTPUT -m owner --pid-owner 3080 -j LOG --log-prefix „A 3080
processz kimeno csomagja: "
```

Sajnos azonban az említett kapcsolók a 2.6-os kernellel már kikerültek a rendszerből, az *owner* modulban már csak az *„—uid-owner", „—gid-owner"* és *„—socket-exists*" kapcsolók léteznek, melyek megfelelő user id, group id vagy socket esetén match-elnek.

Természetesen a régebbi kernel használata nem lehet helyes megoldás, hiszen a 2.6-os kernel óta rengeteg funkcióval, optimalizációval és biztonsági hibajavítással bővült a rendszer. Ezeket elveszíteni felelőtlen döntés lenne.

Processzek futtatása külön felhasználóként

Bár az owner modul *„--pid-owner"* kapcsolója nem használható, az *„--uid-owner"* kapcsoló továbbra is működőképes. Ez azt jelenti, hogy a kimenő csomagokra tudunk olyan szabályt írni, amely a küldő felhasználó user id-jének egyezésekor match-el, vagyis log-olhatjuk a kimenő csomagok küldő felhasználóinak uid-jét. Kissé körülményes megoldással, de ez a kapcsoló segítségünkre lehet processzek adatforgalmának figyelésében is, mindössze annyit kell tennünk, hogy a log-olni kívánt processzeket külön-külön felhasználói fiókokban indítjuk el. Így biztosak lehetünk benne, hogy az adott felhasználó kimenő forgalmát csak a naplózott processz generálja.

Bár ez a módszer működőképes, be kell látnunk, hogy használata több okból kifolyólag is nehézkes. Egyrészt a process id-t így sem kapjuk vissza a log-ban, csak a futtató felhasználót, a pid visszakereséséhez további nyilvántartás szükséges. Másrészt az esetek nagy részében nehezen, vagy egyáltalán nem megoldható a processzek külön felhasználóként történő futtatása: a webszerverek például közös felhasználóban végzik a lekérdezések kiszolgálását.

Netstat

A Linuxban a netstat program alkalmas a hálózati beállítások és hálózati aktivitás figyelésére [4]. *„-tapn”* kapcsolókkal indítva megmutatja a kapcsolatok helyi és távoli portját, valamint a hozzájuk tartozó processzek pid-jét. A netstat sajnos csak a nyitott kapcsolatokat, portokat jeleníti meg, a csomagokról nem kapunk információt, így ez a módszer nem megfelelő a processzek adatforgalmának feltérképezésére, hiszen a nyitott portok száma nincs összefüggésben az azokon át küldött csomagok számára vagy az adatmennyiségre.

4. A kernel hálózatkezelése

A helyi processzektől származó kimenő adatok a hálózatra való kikerülésük előtt hosszú utat járnak be a Linux kernelben. Át kell őket alakítani, csomagokba (socket buffer) rendezni, a csomagoknak pedig a megfelelő header-ökkel kell rendelkezniük, hogy azok az egyes logikai rétegekben értelmezhetőek legyenek. Emellett természetesen a kernel rengeteg dokumentációt, naplózást végez a kimenő csomagokon.

A socket buffer

A Linux kernelben a hálózatkezelés egységeinek nagy részét - így a hálózati csomagokat és a hozzájuk tartozó adatokat is – nagy struktúrákban, úgynevezett socket bufferekben (rövidebben skbuff vagy skb) tárolják. Ezek láncolt listát alkotnak, így épül fel a kernelben például a kimenő, bejövő vagy továbbítandó csomagok sora (socket buffer queue) [2].

Egy buffer 3 nagy részből áll, melyek határait 4 pointer (*head, data, tail, end*) jelöl [6]. A buffer az OSI-modell szerinti adatkapcsolati rétegben (2. szint), hálózati rétegben (3. szint) és szállítási rétegben (4. szint) létezhet. Mivel a kernelben a különböző hálózati rétegekhez tartozó működés jól elkülönül, a csomagokat is meg kell különböztetni annak megfelelően, hogy éppen mely szinten járnak. Így az adatok minden rétegváltáskor új bufferbe kerülnek, amely a régiből klónozással keletkezik [3]. A klónozásért az *__skb_clone()* függvény felelős. A *head* és *tail* egységeknek ilyenkor csak egyes mezői másolódnak, amelyekre a következő rétegben szükség lehet. A bufferben a másolandó mezők kezdetét „headers_start”, végét „headers_end” pointer jelöli. A *data* nem klónozódik, csak egy pointer jön létre ugyanarra a memóriacímre, ahol a régi skb *data* része található.

A socket buffer kódja a */include/linux/skbuff.h* útvonalon érhető el, a header fájlban *struct sk_buff* néven van definiálva.

Socket mechanizmus

A kernel a felhasználói programok számára a Socket API-t biztosítja a hálózati kommunikációhoz. Adatok küldéséhez először létre kell hozni egy kommunikációs végpontot, más néven socketet. Ezt a *socket()* függvénnyel tudjuk megtenni [5].

Protokolltól függően szükség lehet kapcsolat felépítésre is. Az adatok küldésére a Socket API 4 függvényt kínál [1]: *write(), send(), sendto(), sendmsg()*.A függvények mindegyike meghívja a *sendmsg()* virtuális függvényt, melynek segítségével az adatok átkerülnek a szállítási rétegbe [3].

5. A megoldás bemutatása

Mivel a korábban felsorolt áthidaló megoldások nem szolgáltatják az általunk elvárt működést, mindenképpen változtatni kell a Netfilter kódján.

Csomagok társítása processzekhez

Fontos kérdés, hogy honnan tudjuk meg, hogy egy adott csomag melyik processzhez tartozik, azaz melyik folyamat küldte. Mivel a bejövő csomagok csak a Netfilter-en való áthaladásuk után kerülnek a processzek valamelyikéhez, itt nem tudnánk azokat egyértelműen hozzárendelni valamelyik futó programhoz. Ugyanez igaz a továbbítandó csomagokra is.

Ellenben a kimenő csomagokat többnyire egyértelműen folyamatokhoz tudjuk rendelni, hiszen azok nagy részét valamelyik felhasználói program küldi ki a hálózatra és az a processz hamarabb kérte a kerneltől az adatok küldését valamilyen rendszerhívás segítségével, mint ahogy az azokat tartalmazó csomag a Netfilter-hez ér. A függvényhívásokat pedig mind a küldést kezdeményező processz végzi, a kontextus nem változhat. Így az esetek többségében igaz, hogy a csomag küldője az a processz, amely éppen birtokolja a CPU-t, vagyis az aktuálisan futó processz. Ezért többnyire igaz, hogy ha a csomag kernelben történő utazása során lekérdezzük az aktuális processzt, az megegyezik a csomag küldőjével. Az állítás azért csak többnyire igaz, mert néhány szolgálati üzenetet, speciális csomagot maga a kernel küld el, így azokat nem tudjuk felhasználói folyamathoz társítani.

A kernel kódjában létezik egy struktúra, amelyben a kernel az aktuálisan futó processz adatait tárolja. Ez a „current” pointer segítségével elérhető [9] és a processzről minden fontos adatot tartalmaz, amelyek a nyilvántartáshoz szükségesek: sok más információ mellett a PID-et is megtalálhatjuk benne - ezt int típussal a „pid” nevű mező tartalmazza. A *current* struktúra értéke természetesen mindig változik, ha az ütemező másik processznek adja a processzort, a kernel automatikusan kitölti a megfelelő folyamat adataival.

Az sk_buff kibővítése

A csomagot reprezentáló sk_buff struktúra számos mezővel rendelkezik, amelyben típusa szerint a pid eltárolható lenne: elhelyezhetnénk azt például a mark mező értékeként. Ebben az esetben viszont előfordulhat, hogy ha a rendszer másra használja a csomag adott mezőjét, a benne lévő érték később felülíródhat, vagy akár az is, hogy a pid-del írunk felül egy másik modul számára fontos adatot.
Így ha új értéket szeretnénk a socket bufferben tárolni, ahhoz új mezőt kell létrehoznunk. Nem mindegy azonban, hogy az új mezőt az sk_buff mely részéhez fűzöm hozzá. Funkcióját tekintve a *head* rész a legalkalmasabb, viszont ügyelni kell arra is, hogy a mező a buffer klónozásakor azzal együtt klónozódjon, mert nem szeretnénk a pidet elveszíteni a hálózat rétegei közötti váltáskor. A másolandó értékek a *headers_start* és *headers_end* elemek között helyezkednek el, ezért az újonnan létrehozott, process id-k tárolására alkalmas mezőt ebben a zónában hoztam létre, deklarációja a következő:

```
pid_t pid;
```

Ezáltal a struktúra alkalmassá vált a küldő PID-jének tárolására, hiszen az új mező értékét biztosan nem fogják felülírni, a klónozás miatt pedig minden rétegben elérhető lesz, ahol a buffer létezik.

A process ID beírása

A csomag már képes a PID tárolására, de persze arról még gondoskodni kell, hogy bele is íródjon. Korábban vizsgált okok miatt célszerű a hozzárendelést minél hamarabb beállítani, ezért azt a kódrészletet kell módosítani, ahol az skb létrejön. Ez protokollonként és csomagtípusonként változó lehet. Módszerünket a lehető legtöbb eshetőségre fel kell készíteni.
Az skb-be a pid a következő paranccsal kerül be (a létrehozott struktúrát a vizsgált függvények nagyrésze*„skb”* névvel jelöli):

```
skb->pid = current->pid;
```

Ha a szállítási rétegben UDP protokollt használunk, az skb létrehozására az *ip_append_data()* függvény feladata, de ténylegesen az *__ip_append_data()* függvényben történik meg. Mindkét függvény a kernelen belül a */net/ipv4* útvonalon található *ip_output.c* fájlban van definiálva.

Az sk_buff allokálása az *__ip_append_data()alloc_new_skb* részében megy végbe *sock_alloc_send_skb()* hívással. A szükséges hibaellenőrzés kódja után biztosak lehetünk benne, hogy a struktúra sikeresen létrejött, a fentebb bemutatott sor beillesztésével a pid mezőt kitölthetjük a megfelelő értékkel.

TCP protokoll esetén a helyzet valamivel komplikáltabb, mert a buffer előállításáért nem egyetlen kódrészlet felelős. A kapcsolat felépítés státuszának megfelelő típusú csomagok összeállítása külön-külön függvényekben történik. Ezek implementációja összesen 2 fájlban, a kernelen belül */net/ipv4* útvonalon elérhető *tcp.c* és *tcp_output.c* dokumentumokban található.

A TCP adatcsomagok összeállítása a *tcp.c*-ben definiált *tcp_sendmsg()* függvényben zajlik, pontosabban annak *new_segment* részében. Az allokálásra *sk_stream_alloc_skb()* hívódik, majd az UDP-hez hasonlóan itt is a megfelelő hibaellenőrző kódrészletek lefutása után történik a buffer feltöltése adatokkal. A PID beállítására használt kódrészletet ez esetben az adatok betöltése elé iktattam be.

TCP használatakor azonban nem az adatcsomagok elküldése a rendszer egyetlen feladata: fel kell építeni a kapcsolatot, kezelni kell annak állapotát, majd le kell bontani, ha a szükséges kommunikáció befejeződött az állomások között. Ezeket kimenő adatok esetén a *tcp_output.c* fájl függvényei végzik olyan módon, hogy minden speciális TCP üzenetnek külön skb-t generálnak. Ezért a kódot be kell illeszteni a SYN, SYN ACK és ACK csomagokat küldő függvényekbe is a megfelelő helyre, az allokálás után.

Megfigyelhető, hogy a *tcp_connect()* kivételéve ezek mindegyike meghívja a *tcp_transmit_skb()* függvényt a csomag továbbításához. Így alternatív megoldásként a *pid* mező beállítása itt is lehetséges, ha nem ragaszkodunk ahhoz az elvhez, hogy ez azonnal az allokálás után megtörténjen.

Mivel a PID log-olás csak a Netfilter OUTPUT láncán működik, ha az INPUT és FORWARD láncot próbáljuk log-olni, nem lenne szükség a *pid* mező megjelenítésére. A log-ban a megfelelő helyen ekkor 0 értékek tűnnek fel, hiszen a bejövő és továbbítandó csomagokon nem történik meg a *pid* tag beállítása, korábban ismertetett indokok miatt. Ha nem szeretnénk, hogy a felhasználó lássa a 0 értékeket, a *pid* mező ilyenkor egyszerűen eltávolítható a log-ba írt sor végéről egy egyszerű „if” feltétel bevezetésével a kódban.

6. Az implementáció tesztelése a gyakorlatban

A bemutatott módszer hasznosságának jobb megismerése céljából teszteltem azt egy valódi szerveren is. A gépen többek között Apache webszerver is üzemel, amely több website-ot is kiszolgál. Előfordul,

hogy a lefutó PHP scriptek távoli gépekre is szeretnének kapcsolódni. A távoli IP címeket természetesen az IPtables által generált log-okból könnyedén kiolvashatjuk a DST mező alapján. Pl.:

> [71792.592955] webusers.out IN= OUT=eth0 SRC=195.56.148.112 DST=162.243.186.237 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=29140 DF PROTO=TCP SPT=59876 DPT=80 WINDOW=14600 RES=0x00 SYN URGP=0 UID=1075 GID=1075 PID=2200

Az Apache szerver is rendelkezik saját log-gal, amely megadja, hogy melyik kérés kiszolgálása melyik processzben történt: A log utolsó oszlopában látható a folyamat id-je. Pl:

> www.hadas.hu hadas.hu 157.55.39.233 - - [06/Apr/2017:20:18:04 +0200] "GET /index.php/tervek/tervek/tervek/miskolc-zenepalota-felujitasa.html HTTP/1.1" 200 9853 "-" "Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X) AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)" 0 10231344 2200

Alapvetően a két log adatai egymástól függetlenek, a kimenő csomagokról nem tudjuk megmondani, milyen kérés vagy script hatására jöttek létre. Ha viszont figyelembe vesszük az IPtables naplózott sorához újonnan hozzáadott PID mezőt, megfigyelhető, hogy a webszerveren lefutott PHP script processz azonosítója és a kimenő csomag PID mezőjének adata megegyezik. Így megállapítható, hogy a fenti csomagot az Apache logban feltüntetett oldalon lefutó script generálta (ebben a példában a PID 2200). Ezen kapcsolat figyelembevételével tehát meg tudjuk mondani, melyik kimenő csomagot melyik PHP script generálta. Ennek megfelelően az esetleges kártékony vagy tévesen működő scriptek kiszűrhetők, működésük további módszerekkel korlátozható vagy letiltható.

> [71133.905016] webusers.out IN= OUT=eth0 SRC=195.56.148.112 DST=162.243.186.237 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=23204 DF PROTO=TCP SPT=59253 DPT=80 WINDOW=14600 RES=0x00 SYN URGP=0 UID=1075 GID=1075 PID=31791

Az itt látható csomagot például a következő kérés kiszolgálása generálta:

> www.hadas.hu hadas.hu 68.180.228.34 - - [06/Apr/2017:20:07:05 +0200] "GET /index.php/kastelyrekonstrukciok/megvalosult-munkak/kastelyrekonstrukciok/hernadvecse-volt-vecsey-sardagna-kastely.html HTTP/1.1" 200 15310 "-" "Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)" 0 10206796 31791

Ez is bizonyítja, hogy a kimenő csomagokhoz tartozó küldő processz azonosítójának log-olása a webszerverek, weboldalak üzemeltetőinek is segítségére lehet.

7. Összegzés

Dolgozatomban bemutattam, miért lehet fontos a kimenő hálózati csomagok megkülönböztetése küldő processz szerint. Bemutattam a Linux kernel hálózati struktúráját, a létező módszereket a csomagok megfigyelésére, valamint egy általam készített implementációt, melynek segítségével az IPtables LOG target-jének használata során a csomagot küldő processz azonosítója is bekerül a kernel log-ba.

Elmondható, hogy a tárgyalt implementáció segítségével a legtöbb esetben megvalósítható a Linux rendszereken a kimenő csomagokhoz tartozó küldő processzek visszakeresése, azok process id alapján történő naplózása, vagyis a kitűzött célt sikerült elérnem.

Ezáltal újabb információkhoz juthatunk a kimenő csomagokról, a számítógép által generált hálózati forgalomról. A log megfigyelése útján kiszűrhetők a nagy adatmennyiséget küldő processzek és akár a számítógépre vagy adatainkra nézve kártékony szoftverek is: gyanús lehet, ha egy olyan processz küld ki információkat a hálózatra, amelytől egyébként funkcionalitásából adódóan ez nem lenne elvárt. Így a bemutatott módszerrel a kernel még több adatot nyújt az üzemeltetőknek a rendszerrel kapcsolatban. Konkrét példán keresztül is bemutattam, hogyan segít az implementáció az Apache webszerver PHP scriptjeinek megfigyelésében.

A módszer remek tulajdonsága továbbá, hogy megvalósítása egyszerű, a kernel minden egyéb komponensével kompatibilis. Erőforrásigénye rendkívül alacsony, gyakorlatilag elenyésző, így a folyamatos log-olás biztosítható akár gyenge hardvereken is.

Hiányosságok

Sajnos a fenti megoldás nem tér ki minden eshetőségre, ami a log-olás kapcsán felmerülhet. Rendelkezik hiányosságokkal, ezek azonban működését nem nehezítik, további apróbb fejlesztésekkel áthidalhatók. Amint korábban is tárgyaltam, a bejövő és továbbított csomagokhoz tartozó processzek PID-jei a létrehozott módosításokkal nem naplózhatók. Továbbított csomagok esetén ez a hozzárendelés nem értelmezhető. A bejövő láncon sem megállapítható teljes biztonsággal a csomaghoz tartozó processz, becsléseket viszont lehetne rá adni.

Jelen dolgozat ezeket a témaköröket nem fedi le, a célkitűzések között nem szerepelt a módszer további láncokra való kiterjesztése. Ezek lehetőségeinek feltérképezése újabb kutatás témája lehet.

Továbbá a log-olás működéséből adódóan nem minden kimenő csomag esetén kereshető vissza a hozzá tartozó userspace processz. Előfordul például, hogy a TCP protokoll esetén küldött ACK-kat a kernel hozza létre. Ekkor a naplózott PID nem tartozik a felhasználói processzek egyikéhez sem, de logikailag hozzájuk kapcsolódna.

Valamilyen más módszer integrálásával becslést tudnánk adni a rendszer által küldött válaszokhoz tartozó processzek PID-jére is.

Fejlesztési lehetőségek

Bár a PID alapvetően fel van tüntetve a Netfilter által generált log-ban, néhány bővítéssel még kényelmesebbé tehetnénk a felhasználók számára a hálózat terhelésének monitorozását, illetve az esetleges beavatkozást.

Ezek jelen szakdolgozat témájához nem kapcsolódnak szorosan, viszont a feltárt módszer felveti a további bővítések lehetőségét, melyek segítségével akár teljes értékű program is készíthető a kimenő forgalom kezelésére.

Jelen esetben csak a küldő processzek PID-je szerepel a log-ban, így a felhasználó a hozzá tartozó processz nevét, más adatait csak valamilyen más hozzárendelés (pl. *ps* parancs kimenetében generált táblázat) segítségével tudja megállapítani.

Ezeket az adatokat a log-hoz is hozzáfűzhetnénk, mert kiolvasható a processz kontextusból, így nem kellene manuálisan megvizsgálni a küldőt.

A túl sok adatot küldő processzeket egyelőre a kill paranccsal történő bezárással tudjuk gátolni tevékenységükben. Egyes esetekben azonban szükséges lehet egy olyan alternatíva, melyben a folyamatot nem állítjuk le, csupán az általa generált csomagokat dobjuk el.

További fejlesztésként akár a Netfilter használatával is létre lehetne hozni egy programot, amely képes process id alapján engedélyezni vagy eldobni a kimenő csomagokat. A logokból kiolvasott PID-ek alapján már egy egyszerű scripttel is látványos, áttekinthető statisztikát, vagy komolyabb elemzés is készíthető a processzek kimenő forgalmáról.

Köszönetnyilvánítás

Köszönöm konzulensem, Dr. Vincze Dávid egyetemi docens segítségét a kutatás és ezen cikk elkészítése során. Az ő munkája nélkül a bemutatott módszer nem jöhetett volna létre. A cikkben ismertetett kutatómunka az EFOP-3.6.1-16-2016-00011 jelű „Fiatalodó és Megújuló Egyetem – Innovatív Tudásváros – a Miskolci Egyetem intelligens szakosodást szolgáló intézményi fejlesztése" projekt részeként – a Széchenyi 2020 keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

8. Irodalomjegyzék

[1] Rosen, R. (2014): *Linux Kernel Networking – Implementation and Theory,* ISBN: 978-1-4302-6196-4
[2] Lu, H.: *The linux networking architecture* (http://www.slideshare.net/hugolu/the-linux-networking-architecture)
[3] *The Linux Foundation Wiki (Arnout Vandecappelle): Linux kernel flow* (https://wiki.linuxfoundation.org/networking/kernel_flow)
[4] *Linux man page: netstat* (https://linux.die.net/man/8/netstat)
[5] Rosen, R.:*Sockets in the kernel* (http://www.haifux.org/lectures/217/netLec5.pdf)
[6] Vger kernel org.: *How SKBs work* (http://vger.kernel.org/~davem/skb_data.html)
[7] *Netfilter dokumentáció* (http://netfilter.org/documentation)
[8] Russel, R.: *Linux 2.4 Csomagszűrő HOWTO (fordította: Demeter Bence)* (http://szabilinux.hu/iptables/chapter7.html)
[9] Stack overflow: *How does current->pid work for Linux?* (http://stackoverflow.com/questions/10838342/how-does-current-pid-work-for-linux)

A linkek utoljára ellenőrizve: 2018. április 8.

Jelen cikk a szerző engedélyével jelent meg másodközlésben. Az első megjelenés bibliográfiai adatai: Soós Róbert: *Kimenő csomagokhoz tartozó processzek azonosítójának naplózása a Linux kernel netfilter komponensében*. Diáktudomány: A Miskolci Egyetem Tudományos Diákköri Munkáiból 11. pp. 102-108. (2018)